Hierarchical Model to Develop Component-Based Systems

Abdelkrim Amirat and Mourad Oussalah
Laboratoire LINA CNRS FRE 2729, Université de Nantes
2, Rue de la Houssinière, BP 92208,
44322 Nantes Cedex 03, France
{Abdelkrim.Amirat, Mourad.Oussalah} @Univ-Nantes.fr

Abstract

Large and complex software systems require expressive notations for representing their software architecture. In this context Architecture Description Languages (ADLs) can be used for describing architectures of components-based software systems. Typical ADLs provide explicit support for specifying components, connectors, and configuration as well as for building hierarchical systems configurations. All of them allow structural dependencies among components to be specified to define static configurations. This may be sufficient for an initial system composition, but does not provide enough information for reasoning about the different kind of connections among elements. Physical and logical connections are defined in this paper. Four kinds of hierarchies are also presented. Each one is used to provide special-purpose view about the architecture.

1. Introduction

One of the key goals of software architecture research is to understand and to manipulate a system at a higher level of granularity than modules or lines of code. Generally, software architectures are composed of components, connectors and configurations, constraints on the arrangement and behaviour of components and connectors. The architecture of a software system is a model, or abstraction of that system. Software architecture researchers need extensible, flexible architecture descriptions languages (ADLs) and equally clear and flexible mechanisms to manipulate these core elements of the architecture.

Recently Medvidovic [5] gives the following definition for software architecture "*A software system's architecture is the set of design decisions about the system which, if made incorrectly, may cause your project to be cancelled*". However, these design decisions encompass every aspect of the system under development, including:

- Design decisions related to system *structure* – for example, "there should be exactly three components in the system, the *data store*, the *business logic* and the *user interface component*;"
- Design decisions related to behaviour (also referred to as functional) – for example, "data processing, storage, and visualisation will be handled separately;"
- Design decisions related to the system's non functional properties – for example, "the system dependability will be ensured by replicated processing modules;"
- Also, we can elicit other design decisions related to the development process or the business position (*product-line*).

We note that in the description languages architectures (ADLs) that currently exist, there is no standard about architectural concepts or standards in terms of mechanisms for manipulating those concepts (i.g. in the ADLs defining explicit connectors, we see that each one gives its proper definition for connectors, some ADLs define the concept of configuration while others do not).

For the reasoning model, the majority of ADLs proposes only sub-typing (inheritance) as a mechanism for specialization (e.g. Acme, C2). Otherwise, for the rest of ADLs, they propose their own ad hoc mechanisms based on methods designed specially for these ADLs.

Based on a bread survey of architecture description notations and approaches, we identified that ADLs capture aspects of software design centred around a system's *Component*, *connectors*, and *configurations*. The core elements of our model are basically defined around these tree elements. So, from this we derive the name of our model **C3** for **C**omponent, **C**onnector, and

Configuration. Taking into consideration that our C3 have no relationship with C2 defined by Taylor [14] nor with C3 with is an extension of C2 defined by Pérez-Martínez [10].

The rest of the paper is organized as follows. In section 2 presents our research motivations. Section 3 describes the C3 metamodel C3. The last section presents our conclusion and the different perspectives of our work.

2. Motivation

In this work the goal is to develop a generic model for the description of software architectures which must be minimal and complete. It is minimal because we are only interested by the core concepts in each ADL. And complete because with this minimum of concepts the architect can be able to describe any required structures he need to realize.

However, describing only the architecture structure is not sufficient to provide correct and reliable software systems. In this paper we are even more going to focus on representation architecture model and to reason about its elements following four different types of hierarchies. Each of these hierarchies provides a particular view on the architecture. In the following sections we present more details about these hierarchies.

Using our approach software architecture is more explicit and clarified by:

- Make explicit the possible types of hierarchies being used as support of reasoning on the architectures, with the different possible levels in each hierarchy.
- Show semantics convey by every type of hierarchy by providing the necessary mechanisms used to connect elements of in the same level of hierarchy and the mechanisms used to connect elements of every level with the elements of the super and lower levels.
- Allows introducing various mechanisms of reasoning within the same architecture according to the requirement problem in a specific domain space.
- Establish the position of existing mechanisms developed for reasoning with regard to our referential.

3. The metamodel of C3

In order to have a complete C3 model, we define mainly two models to describe and reason about software architectures. A *representation model* to describe any architecture based on C3 elements and a *reasoning model* to understand and analyse the representation model.

3.1. Representation model

The core elements of the C3 representation model are components, connectors, and configurations, each of these elements have an interface to interact with its environment. Figure 1 depicts the core elements of the metamodel C3.

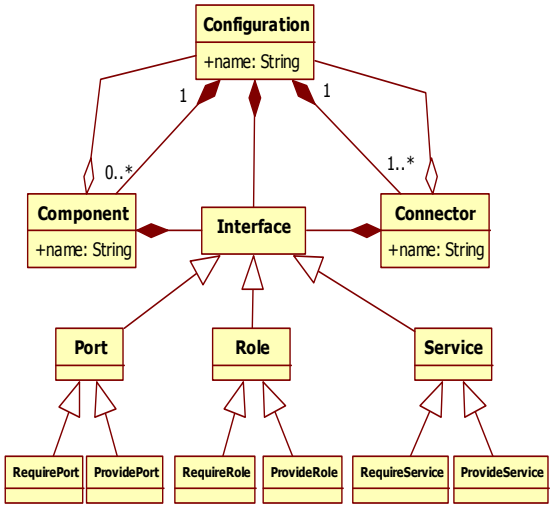


Figure 1. Basic elements of C3 metamodel

3.1.1. Components

A component is a unit of computation or data storage. Therefore, components are loci of computation and state. A component in an architecture may be as small as a single procedure or as entire application. It may require its own data and/or execution space, or it may share them with other components [6].

In order to be able to adequately reason about a component and the architecture that includes it, C3 should also provides facilities for specifying component needs, i.e., services required of other components in the architecture. An interface thus defines computational commitments a component can make and constraints on its usage.

Each interaction in C3 is a *port*. Ports are named and typed. We distinguish between required and

provided ports. Each port is used by one or more services.

Component semantics are modeled to enable evolution, analysis, enforcement of constraints and consistent mappings of architectures from one level of abstraction to another.

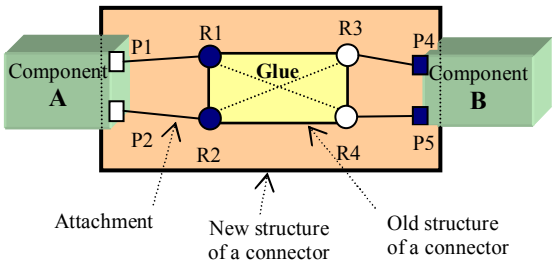
The structure of component is the specification of its required and provided ports. The behaviour of a component is the specification of its required and provided services.

3.1.2. Connectors

Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. In order to enable proper connectivity of components and their communication in an architecture, a connector should export as its interface those services it expects.

C3 refer to connector interaction points as *roles*. Explicit connection (*attachments*) of component ports and connector roles is required in an architecture configuration. Roles are named and typed and are in many ways similar to component ports. Connector services are described inside the glue code [13]. Therefore, a connector's interface is a set of interaction points between it and the components attached to it. It enables reasoning about the well-formedness of an architectural configuration.

Our contribution at this level consists in enhancing the structure connectors by encapsulating the attachment links (Figure 2.a). So, the application builder will have to spend no effort in connecting connectors with its compatible components and configuration. Consequently, the task of the developer consists only in choosing a suitable type of connector which is compatible with the types of components/configurations which are expected to be connected.



Legend: P_i : Port i ; R_j : Role j

Figure 2.a. The new structure of connector

We have given the following definition (Figure 2.b) for connectors in a previous work [1].

Connector_TypeName (*List of element interfaces*)
{
Roles {*List of roles*}
Services {*List of services*}
Properties {*List of properties*}
Constraints {*List of constraints*}
Glue {*The communication protocol*}
Attachments {*List of attachments*}
}

Figure 2.b. Syntax of the connector

So, by encapsulating attachment inside connectors and having well defined connector interfaces with previously known elements to be connected by this connectors, consequently components/configurations and connectors are assembled in an easy and coherent way in the form of an architectural puzzle (*Lego Blocks*) without any effort to describe links among components and connectors or between configurations and connectors. Consequently, this approach accelerates the development of components, improves testability, coherence, maintainability and promotes component markets [1].

3.1.3. Configuration

Architecture configurations, or topologies, are connected graphs of components and connectors that describe architectural structure. This information is needed to determine whether: appropriate components are connected, their interfaces match, connectors enables proper communication, and their combined semantics result in desired behaviour.

The goal of configuration is to abstract away the details of individual components and connectors. They depict the system at a high level that can potentially be understood by people with various levels of technical expertise and familiarity with the problem at hand.

For more clarity, in our model C3 each component or connector is perceived and handled from the outside as primitive element. But their inside can be real primitive elements, or composite with a configuration which encapsulates all the internal elements of this composite. These Configurations are first-class entities. They represent a graph of components and connectors and describe how they are fastened to each other. A configuration may have ports, and each port is bound to one or more ports of the internal components. In general, configurations can be hierarchical where components and connectors represent sub-

configurations that have internal architectures (Figure 2).

3.1.4. Interface

Every architectural element has an interface. Each interface is associated with a type which corresponds to a set of operations which it defines. Via this interface the element publish to the outside environment it needs in term of required services as well as the services which it provides (messages, operations, and variables). However, elements are selected and connected from their published interface. So, the interface is though as a *contract* with the environment that the element should *honour*.

To establish connections between elements we use ports for components and configurations and roles for connectors and we assign the services to each port and role with a necessary set of constraints to be respected during the connections. From conceptual view ports and roles are concrete classes inherited from the interface abstract class as shown in Figure 1.

Also, in modelling level we use cardinality to describe the multiplicity of each relation (*connection*) between architectural elements. This cardinality express the number of ports associated with components and configurations and the number of roles associated with connectors. Each port or role is considered as a channel to carry in/out required/provided services exchanged with element environment.

The previous architectural elements are manipulated and used via predefined mechanisms in the reasoning model. Essentially, we are going to study the instantiation, specialization, composition, decomposition, and connections mechanisms. In the following section, we define the using context of each mechanism. Some details about the structure of these architectural elements are presented in our previous works [1] and [9].

3.2. Reasoning model

In our approach we intended to analyze the software architecture by using different hierarchies where each hierarchy is investigated at different levels of representations. The Figure 3 illustrates the C3 reasoning model of C3. This model is defined by four types of hierarchies and each type represents a specific view on the C3 representation model different from the others. In the following sections, we present those types of hierarchy and we investigate the possible levels of each hierarchy.

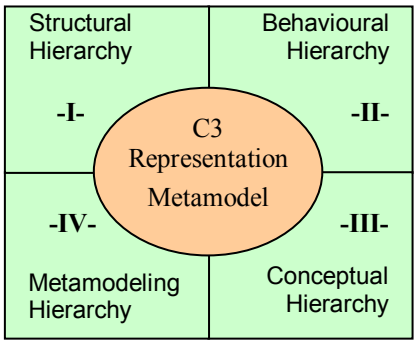


Figure 3. C3 metamodel and reasoning models

In Figure 3 we represent the different reasoning hierarchies based on an external view (*perceived by user*). In the following sections we present: 1- The abstraction hierarchy used to explicit the different nested levels of structural hierarchies that software architecture can have. 2- The behavioural description hierarchy to explicit the different levels of system behaviour hierarchies represented by protocols. 3- The conceptual hierarchy to describe the library of element types at the architecture level. 4- The metamodeling hierarchy to locate our model in the pyramid of hierarchies defined in our previous work. All those external views are associated with an internal views generated automatically using the reasoning tools [1].

3.2.1. Structural hierarchy (SH)

Structural hierarchy also called abstraction hierarchy has to provide the structure of a particular architecture in terms of the architectural elements defined by the ADL. The majority of academic ADLs like Aesop, MetaH, Rapide, SADL, and others [4] or the industrials like CORBA, CCM/CORBA, EJB/J2EE [11] allow only a flat description of software architectures.

Using those ADLs architecture is described only in terms of components connected by connectors without any nested elements - without any structural hierarchy. This design choice was made in order to simplify the structure and also by lack of concepts and mechanisms that respectively define and manipulate configurations of components and connectors.

In our C3 model the structure of architectures is described using components, and connectors, and configurations where configurations are composite elements. Each element in this configuration (component or connector) can be a primitive (with a basic behaviour scenario) or configuration which

contains another set of components and connectors, which in their turn can be primitive or composite material, and so on. However, the metamodel C3 allows the representation of architecture with a real hierarchy (*with an arbitrary n abstraction levels*). It should be noted that practically all architectural solutions for domain problems, have a nested hierarchical nature. Thus, software architecture can be viewed as a graph where each internal node of this graph represents a configuration and each end-node represents a primitive component arcs between nodes are connectors.

In Figure 4.a, we represent the logical view of the structural hierarchy where the root node is the first level of abstraction; it is also the configuration which encapsulates all elements of the architecture. The small white circles represent primitive components and small black circles represent sub-configurations (composites) in the system architecture. These configurations contain other elements inside. Thus, the configurations will never be end-nodes in the hierarchy tree of abstractions. The node with double circles represents the global configuration of the architecture. The arcs represent the bonds of hierarchy - the father/child relationship. This relationship does not necessarily imply a service-connection between the father node and the child one. To navigate among abstraction hierarchy levels we define the following type of connector:

- *Composition-Decomposition Abstraction Connector* (CDAC) used to link each configuration to its underling elements. Therefore, this type of connector allows the navigation among levels of the abstraction hierarchy. Also, we can determine the children or the father, if it is the case, of each element deployed in the architecture. Figure 4.b represents the notation adopted.

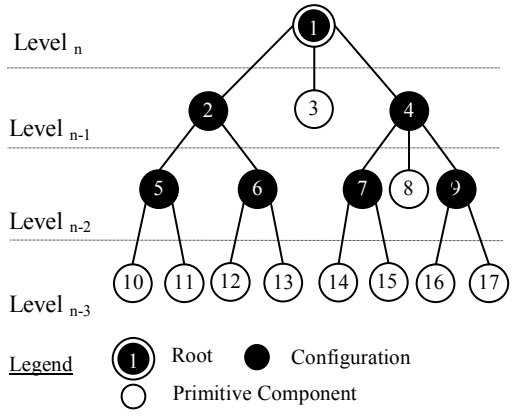


Figure 4.a. Logical view of the SH (1)

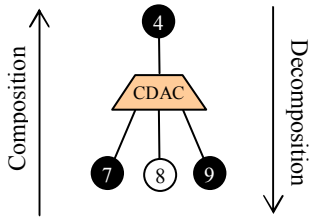


Figure 4.b. CDAC connector

Figures 5.a illustrates the physical view of the structural hierarchy with two service-connection types of connector; the fist one is generally represented by an implicit link called *Binding* and the second one is defined by several ADLs as *Attachment* link. In our model those two types of link are explicated as first class entities and are defined as follows:

- *Expansion-Compression Connector* (ECC) is represented by discontinuous arc. We use this type of connectors to establish service-connections between each configuration and the underling elements (Figure 5.b). In some ADLs this type of link is called *binding or delegation* but not defined as a first class entity.

- *Structural Attachment Connector* (SAC) is represented by full arc. We use this type of connectors to establish service-connections between components and configurations deployed in the same level of abstraction (Figure 5.c). In some ADLs this type of connector in called *assembling connector* and represented by first class entity (e.g. Acme) [2].

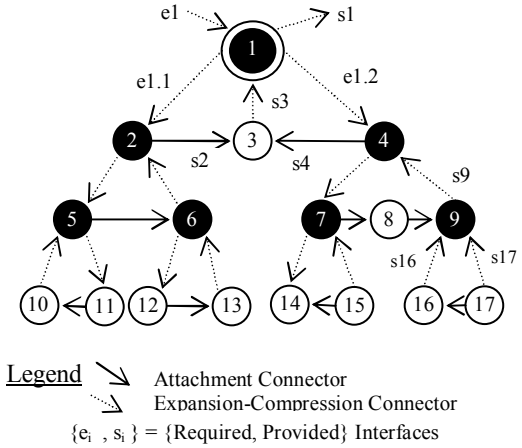


Figure 5.a. Physical view of the SH (2)

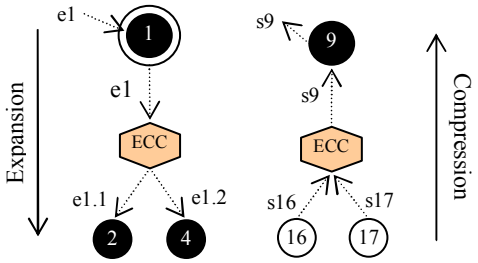


Figure 5.b. ECC connector

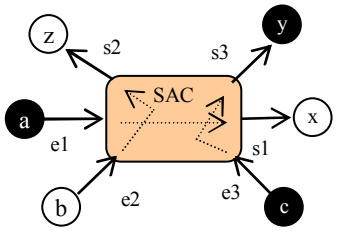


Figure 5.c. SAC connector

Inside the connector the glue define the mapping between elements.

- The provided service (s1) of "a" is required by "x"
- The provided service (s2) of "b" is required by "z"
- The provided service (s3) of "c" is required by "y"

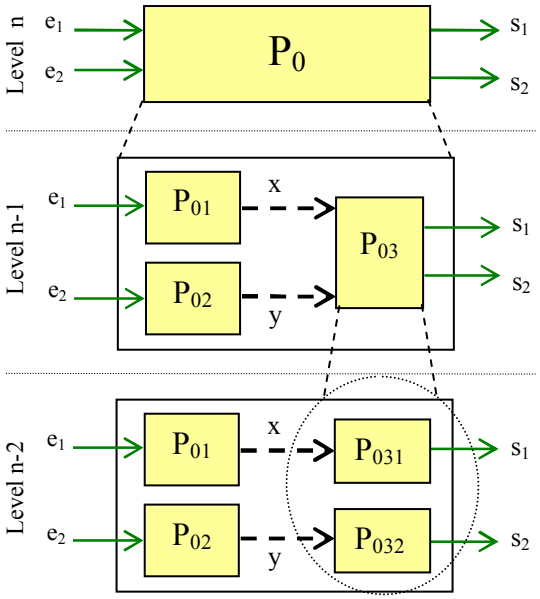
Attachments are also defined inside the connector.

The different elements of the architecture are connected through their interfaces. Thus the types of interfaces are checked if they are compatible or not (*interface matching*). Consequently, in the abstraction hierarchy, the consistency of elements assembly is controlled syntactically.

3.2.2. Behavioural description hierarchy

The behavioural description hierarchy represents the description of the system's behaviour at different levels. Each primitive element of the architecture has its own behaviour. The behaviour description associated with the highest level of the hierarchy - level n in Figure 6.a - represents the overall behaviour of the architecture. This behaviour is described by a global protocol P_0. The system architecture at this level is perceived as a black box with inputs (*required services*) and outputs (*provided services*). At lower level each component, connector, configuration, port, or role has its own protocol to describe its functionality (e.g. glue code is the protocol describing the connector behaviour, also the component behaviour can be described by a state chart diagram). So, protocol is a mechanism used specifying the behaviour of an architectural element by defining the relationship among the possible states of this element and its ability to produce coherent results.

Figure 6.a sketches how to decompose the protocol P_0 at level n into its sub-protocols at level n-1. This decomposition process produces a set of other behaviours $\{P_{01}, P_{02}, P_{03}\}$. By the same process each protocol of the level n-1 is decomposed to produce an other set of sub-protocols at the level n-2, and so on until level 0. The last level of the hierarchy is a set of protocols representing the behaviour of primitive elements available in the library of the architect. The total set of protocol levels represents the behaviour hierarchy of the system architecture.



Legend: P_i: Protocol i ; e_i : Input i ; s_i : Output i ; x,y : intermediate result

Figure 6.a. Logical view of behavioural hierarchy

To navigate among behavioural description hierarchy levels we define the following type of connector:

- *Composition-Decomposition Behavioural Connector* (CDBC) used to link each protocol to its possible sub-protocols. Therefore, this type of connector allows the navigation among levels of the behavioural description hierarchy. Also, we can

determine the children or the father, if it is the case, of each protocol used in the architecture. Figure 6.b represents the notation adopted.

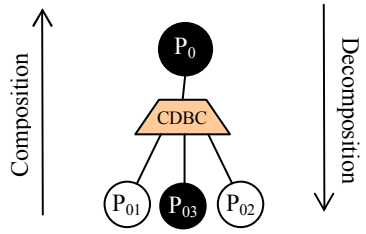


Figure 6.b. CDBC connector

- *Binding Identity Connector* (BIC) used to keep the identity and the traceability of inputs and outputs of protocols. There is no expansion or compression of respectively inputs and outputs of protocols like in abstraction hierarchy. The identity of inputs and output is preserved (Figure 6.c).

- *Behaviour Attachment connector* (BAC) used to connect protocols belonging to the same level of hierarchy. This connection is explicated by real transition between the end-state of the first protocol and the start-state of the second one (Figure 6.c)

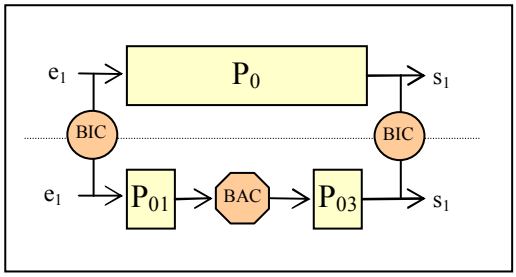


Figure 6.c. BIC and BAC connectors

If we use, for example, transition-based system to specify the behaviour protocol associated with each element then connections between behaviours are made by simple transitions between the end-state of the fist protocol and the start-state of the second one. Inputs and outputs of each protocol are respectively required and provided services.

The syntactic correction (discussed before) of the assembled elements cannot insure the validation of the produced architecture. The syntactic correction checks only the compatibility of interfaces types. So, elements are compatible to exchange information, but fail to check if their collaboration "*the semantic of connections*" can produce a coherent result. Consequently, the behavioural description can insure the compatibility of protocols (*protocol matching*) associated with elements at any level of the hierarchy [3].

3.2.3. Conceptual hierarchy

The conceptual hierarchy allows the architect to model the relationship among elements of the same family as illustrated in Figure 7.a. The architectural entities are represented by types (classes). Each type is a class library and each class has its sub-classes in the library. So, we can shape the graph representing entities hierarchy of the same family. Each graph has its proper number of levels (sub-type levels).

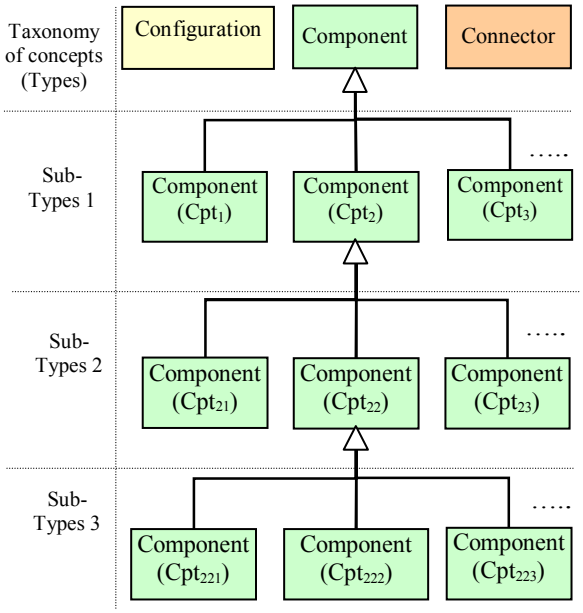


Figure 7.a. Logic view of the conceptual hierarchy

At the highest level of hierarchy we have the basic element types developed to be reused. The element types of the intermediate levels are created by reusing the previous ones. Those intermediate element types are reused to produce others (*development by reuse and to be reused*) or used as end-elements to describe architectures, and so on. Element types at the last level of the hierarchy are only created to be used in the description of architectures.

Through the mechanism of specialization (*inheritance*) the architect will classify the library of elements according to architecture development needs

in each target domain. The number of sub-class levels is unlimited. But we must remain at reasonable levels of specialization in order to keep compromise between the use and the reuse of the architectural elements. To navigate among levels of the conceptual hierarchy we define the following type of connector:

- *Specialisation-Generalisation Conceptual Connector* (CGCC) is used to connect element types coming from the same type (implemented by inheritance mechanism in Java). So, we can construct easily all trees representing the classification library types. Figure 7.b represents the notation adopted.

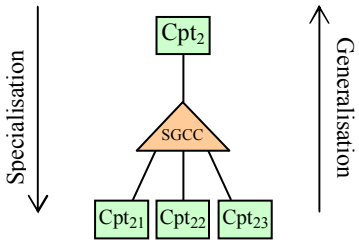


Figure 7.b. SGCC connector

3.2.4. Metamodeling hiérarchy

In the metamodeling hierarchy of we have only 4 architectural levels with instantiation mechanism. Thus, according to Figure 8.a, each level (A_i) must conforms with the description given above in $A_{(i+1)}$ level (*instance-of relationship*). The level A3 conforms to itself. Symmetrically, each level (A_i) describe the inferior level $A_{(i-1)}$. A_0 is the end-level (run-time instance) [7] [8].

A_0 Level is the real word level (*application level*) which is an instance of the architecture model level A_1. At this level the developer has the possibility to select and instantiate elements any times as he needed to describe a complete application. Instances are created from element types which are defined at A_1. Elements are created assembled with respect to the different constraints defined at A_1.

A_1 Level is also called *architecture level*. At this level we have models of architecture, possibly with a given style, described using the language constructions or notations defined at A_2 level (e.g. C3 metamodel). Thus, each architecture model is an instance of the metamodel defined in the above level.

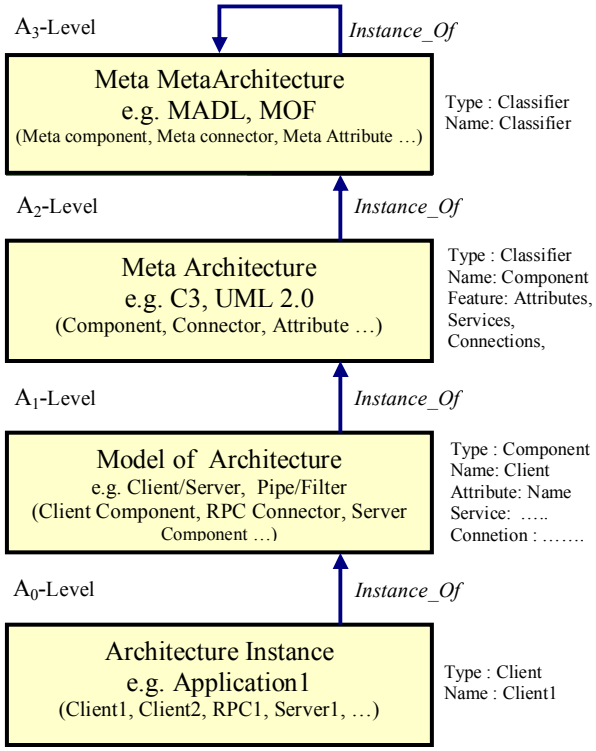


Figure 8.a. External view of the metamodeling hierarchy

A_2 Level (*meta-architecture level*) defines the language or the notation used to describe architectures at A1. This level is also used to modify or adapt the description language. All operations are undertaken at this level will always be in conformance with the last level.

A_3 Level (*meta meta-architecture*) has the top level concepts and elements used when we want to define any new architecture description language or new notation. In our previous work we have defined our proper meta meta-architecture model called MADL. So, our C3 metamodel is defined in conformance with MADL. MADL is similar to MOF but component-oriented [12].

To connect architecture levels we define the following connector:

- *Instance-Of Connector* (IOC) is used to establish connection among elements of a given level (*model*) with their classifier defined in the above level (*metamodel*). Figure 8.b represents the notation adopted.

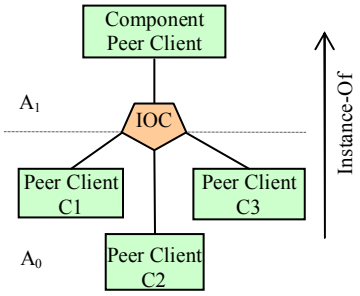


Figure 8.b. IOC connector

5. Conclusion

The success of the component-construction paradigm in mechanical and electrical engineering has led to call its adoption in software development. To this end we have defined a minimal and a complete representation metamodel called C3 to describe software architecture and to reason about this architecture from different perspective view. The core elements of C3 are components, connectors and configurations. Elements are assembled using their interfaces. Syntactic and semantic corrections are carried out using respectively interfaces-matching and protocols-matching. Perspective views are defined by different kind hierarchies. Mainly, we use abstraction hierarchy to describe the structural decomposition hierarchy, behaviour description hierarchy to describe the behaviour decomposition, conceptual hierarchy to describe sub-classes of architectural elements and finally the metamodeling hierarchy to show how we can modify the metamodel C3 and how to use it. Each hierarchy is supported and tooled by explicit connection mechanisms to provide the different form of connections required in each hierarchy.

6. References

[1] A. Amirat, M. Oussalah, and T. Khammaci, "Towards an Approach for Building Reliable Architectures", *Proceedings of IEEE IRI'07*, Las Vegas, Nevada, USA, August 2007, pp 467-472.

[2] D. Garlan, R.T. Monroe, and D. Wile, "Acme: Architectural Description Component-Based Systems, Foundations of Component-Based Systems". *Cambridge University Press*, 2000, pp. 47-68.

[3] A. Lanoix, D. Hatebur, M. Heisel, and J. Souquières, "Enhancing Dependability of Component-Based Systems", *Proceedings of Ada-Europe*, 2007, pp. 41-54.

[4] J. Matevska-Meyer, W. Hasselbring, and R. Reussner, "Software architecture description supporting component deployment and system runtime reconfiguration", *Proceedings of Workshop on Component-Oriented Programming WCOP'04*, Oslo, Norway, June 2004.

[5] N. Medvidovic, E. Dashofy, and R.N. Taylor, "Moving Architectural Description from Under the Technology Lamppost", *Information and Software Technology Journal,* Vol. 49, No. 1, 2007, pp.12-31.

[6] N. Medvidovic, "Architecture-Based Specification-Time Software Evolution", PhD Thesis, University of California, Irvine, 1999.

[7] OMG: "Unified Modeling Superstructure" [Electronic Version] from http://www.omg.org/docs/ptc/06-04-02.pdf, 2006.

[8] OMG: "Unified Modeling Language: Infrastructure" [Electronic Version] from http://www.omg.org/docs/formal/07-02-06.pdf, 2007.

[9] M. Oussalah, A. Amirat, and T. Khammaci, "Software Architecture Based Connection Manager", In Proceedings of SEDE'07, Las Vegas, Nevada, USA, July 2007, pp.194-199.

[10] J.E. Pérez-Martínez, "Heavyweight extensions to the UML metamodel to describe the C3 architectural style", *ACM SIGSOFT Software Engineering Notes*, Vol.28 No.3, May 2003.

[11] M. Pinto, L. Fluentes, and M. Troya, "A Dynamic Component and Aspect-Oriented Platform", *The Computer Journal,* Vol.48 No. 4, 2005, pp. 401-420.

[12] A. Smeda, M. Oussalah, and T. Khammaci, "MADL: Meta Architecture Description Language", *Proceeding of the 3rd ICIS International conference on Software Engineering Research, Management & Applications, SERA'05*, Pleasant, Michigan, USA, August 2005, pp. 152-159.

[13] A. Smeda, M. Oussalah, and T. Khammaci, "Improving Component-Based Software Architecture by Separating Computations from Interactions", *First International Workshop on Coordination and Adaptation Techniques for Software Entities, WCAT'04*, Oslo, Norway, 2004.

[14] R.N. Taylor, N. Medvidovic, K.M. Anderson, JR. E.J. Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow, "A component- and message-based architectural style for GUI software", *IEEE Transaction Software Engineering,* Vol. 22, No. 6, June, 1996, pp.390–406.